Rapport : Projet de Programmation 1

Partie 1 : Questions

Question 1

Il est possible, lors d'un unique parcours de l'arbre de syntaxe abstraite, de stocker dans un buffer l'ensemble des chaîne de caractère à envoyer sur la sortie standard. Une fois le parcours terminé il suffit de placer en premier dans le buffer, les ligne de code correspondants aux déclarations des chaines de caractère puis d'envoyer le contenu du buffer sur la sortie standard. Ce n'est cependant pas la méthode que j'ai choisi.

Question 2

Le mot clé *let* permet de déclarer une association entre un nom et une expression d'un certain type. Si ce type ne comporte pas le mot clé *ref*, cette association peut être vue comme un élément de sucre syntaxique permettant d'écrire des programmes plus concis.

L'ajout du mot clé *ref* signifie que l'on déclare une variable au même titre qu'une variable en C. Une variable est une case de mémoire unique dont le contenu peut être mis à jour partout où la variable est visible.

Une fonction d'un programme Ocaml utilisant des variables génère donc des effets de bord difficiles à détecter. Une fonction n'utilisant pas de variable est entièrement spécifiée et ne produit pas d'effet de bord. De plus si une variable est globale à un fichier, elle peut entrer en conflit de nom avec une autre variable globale d'un autre fichier. C'est pourquoi il est préférable de ne pas utiliser les variable dans les programmes Ocaml.

Question 3

Afin de déclarer les chaines de caractères, j'effectue un premier parcours de l'arbre de syntaxe abstraite. Lors de ce parcours, je déclare également les noms des exceptions présentent après les mots clés *catch* et *throw*, comme des chaînes de caractère et je les rajoutes dans l'environnement, à la liste des chaines de caractère. Lors de la compilation du mot clé *throw*, j'ajoute l'adresse du nom de l'exception lancée dans le registre callee-save *r14*. Ainsi, lors de la compilation d'un *try*, pour chacune des clauses *catch*, je récupère l'adresse du nom de l'exception à attraper et je la compare avec l'adresse stockée dans le registre *r14*. Ainsi si les adresses sont identiques, elles pointent sur la même chaîne de caractère.

Question 4

La clause *finally* est presque toujours exécutée, si le *try* correspondant est exécuté. la seule exception est si un *return* est exécuté dans un bloc catch appartement au *try* correspondant. Dans le cas d'un *return* dans le bloc *try*, le bloc *finally* est exécuté. Dans cet exemple, la clause *finally* n'est pas exécutée :

```
try {
    throw A 5;
} catch (A a) {
    return 4;
} finally {
    printf("Ce message ne s'affichera pas");
}
```

Question 5

Hormis l'ajout de nouveaux éléments à mon environnement, et la déclaration des exceptions dans la liste des chaines de caractères, seul le code écrit pour la compilation du *CRETURN* et du *CALL* a été modifié.

Pour le *CRETURN* : Si lors de l'exécution du premier bloc d'un *try*, si l'on tombe sur un *return* et si la clause *try* comporte un bloc *finally*, ce dernier bloc doit absolument être exécuté avant le retour de la fonction.

Pour le *CALL* : Juste après l'appel d'une fonction, il faut vérifier si une exception a été levée. Pour cela, j'utilise une variable globale assembleur *.exception_raised*, initialisée à 0, et mise à 1 lors de l'exécution d'un *throw*. Ainsi, après l'appel d'une fonction je test si *.exception_raised* vaut 1, et si la fonction est appelée dans un bloc *try*, auquel cas je cherche la clause *catch* correspondante. Sinon j'exécute la sortie immédiate de la fonction.

Il serai possible, lors de la compilation du *CTRY* de faire un parcours spécifique de l'arbre de syntaxe à la recherche de *return* et d'appels de fonction. Cela est cependant extrêmement lourd et amène et réécrire du code déjà écrit.

Question 6

L'utilisation des exceptions permet une gestion plus fine des erreurs et ainsi permet d'éviter de nombreux plantages d'un programme. En effet dans une situation ou un code peut potentiellement générer une erreur, il suffit d'entourer ce code d'une clause *try* pour ainsi traiter l'erreur potentielle et continuer l'exécution du programme sans pour autant que ce dernier ne plante. La clause *finally* permet par exemple en cas d'erreur irrattrapable, d'effectuer une ultime action comme la fermeture ou la sauvegarde d'un fichier en cours d'utilisation, ou encore l'enregistrement dans un fichier de log ou l'envoie vers un serveur, des informations concernant l'erreur.

Partie 2 : Compilation des exceptions

L'environnement

L'environnement Ocaml comporte maintenant les variables supplémentaires suivantes :

```
type env = {
  ...
  catch_label : string;
  depth_try : int;
  finally_labels : string list;
  in_finally : bool;
}
```

- catch_label : Lors de la compilation du code d'un bloc *try*, on génère un label assembleur vers lequel sauter pour arriver devant un code qui effectuera les comparaisons entre l'exception levée et les différentes exceptions qui peuvent potentiellement être attrapées. Ce label est stocké dans la variable *catch_label*.
- depth_try : il s'agit de la profondeur du bloc *try* courant dans les imbrications de blocs *try*. Voici un exemple montrant la valeur de *depth_try* lors de sa compilation,

```
depth = 0
try {
    depth = 1
    try {
        depth = 2
    } catch(E e) {
        depth = 1
    } finally {
        depth = 1
    }
    depth = 1
} catch(E e) {
    depth = 0
} finally {
    depth = 0
}
depth = 0
```

Cela permet de vérifier si l'on se trouve dans un *try*, afin de vérifier si une exception peux être attrapée, directement après un *throw* ou après un

appel de fonction

- finally_labels : Lors de la compilation d'un *try*, un label assembleur est généré et correspond à l'endroit où sauter afin d'aller à l'exécution de la clause *finally*. Ce label est ajouté en tête de la liste *finally_labels* qui contient tous les labels de tous les blocs *finally* des clauses *try* courantes imbriquées. Voici un exemple montrant un code et la valeur courante de *finally_labels* lors de sa compilation, en supposant que finally_label_n est un label qui permet d'effectuer le finally n

```
finally_labels = []
try {
    finally_labels = [finally_label_2]
    try {
          finally_labels = [finally_label_1; finally_label_2]
    } catch {
          finally_labels = [finally_label_1; finally_label_2]
    } finally {
          //finally 1
          finally_labels = [finally_label_2]
    }
} catch {
    finally_labels = [finally_label_2]
} finally {
    //finally 2
    finally_labels = []
}
finally_labels = []
```

Cela permet, lors d'un *return* dans un *try* lui même imbriqué dans d'autres *try*, d'effectuer toutes les clauses *finally* des différents *try*

- in_finally : Un booléen indiquant si le code couramment exécuté se trouve dans une clause *finally* ou non.

L'environnement assembleur comporte les deux variables globales suivantes, déclarées et initialisées à 0 au début de la compilation d'un programme :

- .exception_raised : cette variable est mise à 1 après l'éxecution d'un *throw*. Elle restera à 1 jusqu'à la gestion de l'exception levée.
- .return_label_set : cette variable est mise à 1 juste avant l'exécution d'un *return* se trouvant dans un bloc *try*. Elle permet à ce que juste après l'exécution d'un *finally* n'atteignant pas de *return*, de savoir s'il reste un *return* à exécuter dans le bloc *try* correspondant. Si c'est le cas, un label aura été stocké dans le registre *r13* qui est callee-save. Il suffit de sauter vers ce label en utilisant l'étoile :

```
jmp   *%r13
```

De plus, la valeur à retourner est stockée dans le registre callee-save *r15*. En effet, si c'est une variable qui est retournée, le bloc *finally* peut potentiellement la modifier alors qu'il faut retourner sa valeur au moment de l'arrivée sur le *return*

Stratégie de compilation

La première étape consiste à déclarer et initialiser les deux variables assembleur présentées plus haut. Ensuite, les noms des exceptions sont déclarés comme des chaînes de caractères et stockés dans l'environnement, comme expliqué dans la réponse à la question 3.

- Compilation de CTHROW :
  - La variable *.exception_raised* est mise à 1
  - Le nom de l'exception est cherchée dans l'environnement afin de récupérer l'adresse de sa chaîne de caractère qui est ensuite stockée dans le registre callee-save *r12*
  - L'expression associée au *throw* est exécutée et sa valeur est stockée dans le registre callee-save *r14*
  - Si le *throw* se trouve dans un bloc *try*, on saute directement vers le label stocké dans la variable *env.catch_label*. On peut vérifier cela grâce à la variable *env.depth_try*.
  - Sinon on quitte la fonction avec *leave* et *ret*

.

- Compilation de CRETURN :
  - Si on se trouve dans un bloc *finally*, on remet 0 dans *.exception_raised* et *.return_label_set*. En effet, dans ce cas on quitte la fonction normalement, toutes les exceptions ont été gérées
  - On exécute l'expression à retourner et on stocke sa valeur dans le registre *r15*
  - Si on se trouve dans un bloc *try*, et que ce *try* comporte un bloc *finally* (c'est le cas si *env.finally_labels* est différent de la liste vide), pour chacun des labels dans *env.finally_labels*, on effectue les opérations suivantes.
  - On génère un label de retour, *return_label* que l'on stocke dans le registre *r13*
  - on met la variable *.return_label_set* à 1.
  - On exécute le bloc *finally* correspondant en sautant sur le label
  - On produit le code `java return_label:` Ainsi les différents bloc *finally* vont être éxectutés. Si l'un d'eux comporte un *return* les autres seront éxectués de manière récursive et c'est la valeur du bloc *finnaly* contenant un *return* le plus à l'extérieur qui sera renvoyée.
  - Si on a sauvegardé une valeur dans *r15*, on retourne cette valeur. Sinon on compile l'expression *return* comme dans la première partie du projet

.

- Compilation du CTRY :
  - On génère un label *label_end* permettant de sauter à la fin des catchs
  - S'il y a un bloc *fianlly*, on génère un label permettant de sauter au *finally*, que l'on ajoute en tête de la liste *finally_labels*
  - On incrémente la variable *env.depth_try* de 1
  - On génère une liste de labels pour chacun des catchs afin de sauter vers les codes correspondants si une exception levée est attrapée
  - On stocke le contenu de la variable *env.catch_label* et l'on génère un nouveau label permettant d'accéder aux comparaisons entre la listes des exceptions attrapées et l'exception courante levée. On stocke ce label dans la variable *env.catch_label*
  - On compile le code du bloc *try* puis on restaure la variable *env.catch_label*
  - On produit le code effectuant les comparaisons comme expliqué dans la réponse à la question 3
  - Pour chacun des catchs, on commence par produire le code :

```
label:
```

    Ou *label* est le label généré plus haut, et correspondant au catch
  - Pour une clause (*catch E e*) On alloue sur la pile pour ce catch uniquement, une variable locale de nom *e* où l'on met la valeur du registre *r12*
  - On compile le code correspondant et on saute sur le label de fin générer en première étape
  - On restaure l'ancienne valeur de *env.finally_labels* en supprimant son premier élement Les dernières étapes ne s'effectuent que dans le cas où il y a un bloc *finally*
  - On compile le code associé en passant dans l'environnement la variable *env.in_finally* à vrai
  - Si l'on arrive à la fin, c'est à dire qu'aucun *return* n'a été rencontré : On vérifie la présence d'un *return* rencontré dans le bloc *try* grâce à la variable *.return_label_set*. Si cette variable vaut 1, on exécute le *return* comme expliqué plus haut
  - Sinon on vérifie si une exception a été levée dans le bloc *finally* auquel cas on quitte la fonction avec *leave* et *ret*

.

- Compilation du CALL
  - Après avoir éxecuté l'instruction *call*, on vérifie si la variable *.exception_raised* est à 1 auquel cas une exception a été levée dans la fonction Si c'est le cas,
    - Si on se trouve dans un bloc *try*, on saute sur le label stocké dans *env.catch_label*
    - Si on ne se trouve pas dans un bloc *finally* on quitte la fonction

(en effet dans le cas d'un bloc *finally* il faut continuer et terminer l'exécution du bloc)

Le reste du code n'est pas modifié.

Avantages et inconvéniants

- L'utilisation d'un type enregistrement permet d'avoir un unique argument *env* à passer en paramètre des fonctions. Cependant une modification d'un champ de l'environnement nécessite de faire une copie de celui-ci, ce qui engendre une syntaxe lourde
- Maintenir dans l'environnement la liste des labels vers les différents bloc *finnaly* des clauses *try* courantes est également très lourd. Pour gérer les cas de clauses *try* imbriqués_, il existe surement des façons plus naturelles de respecter la sémantique pour gérer les cas de clauses *try* imbriqués. En pensant de manière plus récursive notemment.
